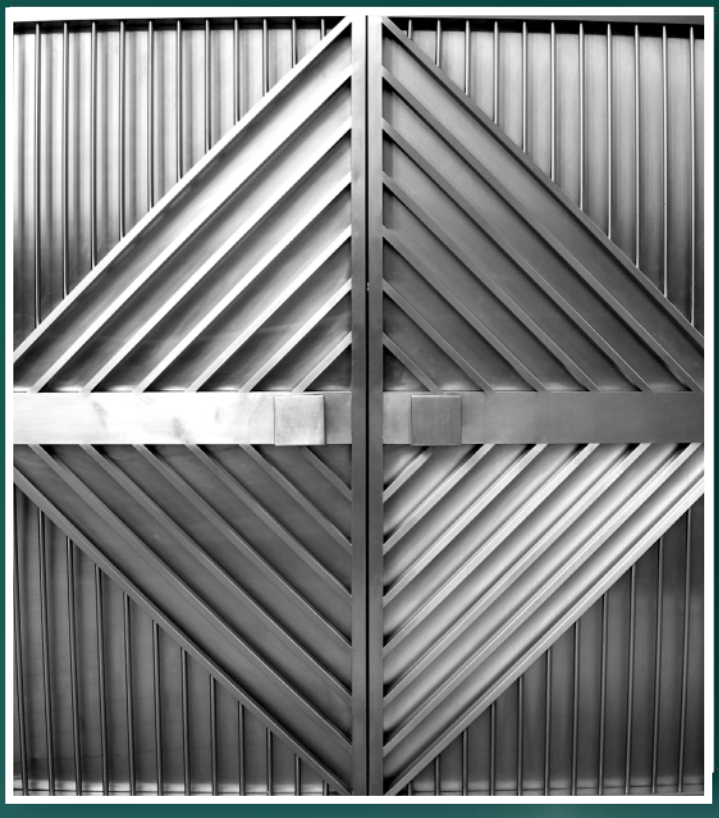


Chapter 10

Service API and Contract Versioning with Web Services and REST Services

NOTE

This chapter provides a number of code examples that help demonstrate various versioning scenarios and approaches. Note that these code examples are not related to any code examples provided in Case Study Examples from preceding chapters.

After a service contract is deployed, consumer programs will naturally begin forming dependencies on it. When we are subsequently forced to make changes to the contract, we need to figure out:

- Whether the changes will negatively impact existing (and potentially future) servicec onsumers
- How changes that will and will not impact consumers should be implemented and communicated

These issues result in the need for versioning. Anytime you introduce the concept of versioning into an SOA project, a number of questions will likely be raised, for example:

- What exactly constitutes a new version of a service contract? What's the difference between a major and minor version?
- What do the parts of a version number indicate?
- Will the new version of the contract still work with existing consumers that were designed for the old contract version?
- Will the current version of the contract work with new consumers that may have different data exchange requirements?
- What is the best way to add changes to existing contracts while minimizing the impact on consumers?
- Will we need to host old and new contracts at the same time? If yes, for how long?

We will address these questions and provide a set of options for solving common versioning problems. The upcoming sections begin by covering some basic concepts, terminology, and strategies specific to service contract versioning.

10.1 Versioning Basics

So when we say that we're creating a new version of a service contract, what exactly are we referring to? The following sections explain some fundamental terms and concepts and further distinguish between Web service contracts and REST service contracts.

Versioning Web Services

As we've established many times in this book, a Web service contract can be comprised of several individual documents and definitions that are linked and assembled together to form a complete technical interface.

For example, a given Web service contract can consist of:

- One (sometimes more) WSDL definitions
- One (usually more) XML Schema definitions
- Some (sometimes no) WS-Policy definitions

Furthermore, each of these definition documents can be shared by other Web service contracts. For example,

- A centralized XML Schema definition will commonly be used by multiple WSDL definitions.
- A centralized WS-Policy definition will commonly be applied to multiple WSDL definitions.
- An abstract WSDL description can be imported by multiple concrete WSDL descriptions or vice versa.

Of all the different parts of a Web service contract, the part that establishes the fundamental technical interface is the abstract description of the WSDL definition. This represents the core of a Web service contract and is then further extended and detailed through schema definitions, policy definitions, and one or more concrete WSDL descriptions.

When we need to create a new version of a Web service contract, we can therefore assume that there has been a change in the abstract WSDL description or one of the contract documents that relates to the abstract WSDL description. The Web service contract content commonly subject to change is the XML schema content that provides the types for the abstract description's message definitions. Finally, the one other contract-related

technology that can still impose versioning requirements but is less likely to do so simply because it is a less common part of Web service contracts is WS-Policy.

Versioning REST Services

If we follow the REST model of using a uniform contract to express service capabilities, the sharing of definition documents between service contracts is even clearer. For example,

- All HTTP methods used in contracts are standard across the architecture.
- XML Schema definitions are standard, as they are wrapped up in general media types.
- The identifier syntax for lightweight service endpoints (known as resources) are standard across the architecture.

Changes to the uniform contract facets that underlie each service contract can impact any REST service in the service inventory.

Fine and Coarse-Grained Constraints

Regardless of whether XML schemas are used with Web services or REST services, versioning changes are often tied to the increase or reduction of the quantity or granularity of constraints expressed in the schema definition. Therefore, let's briefly recap the meaning of the term *constraint granularity* in relation to a type definition.

Note the bolded and italicized parts in Example 10.1:

```
<xsd:element name="LineItem" type="LineItemType"/>
<xsd:complexType name="LineItemType">
  <xsd:sequence>
    <xsd:element name="productID" type="xsd:string"/>
    <xsd:element name="productName" type="xsd:string"/>
    <xsd:any minOccurs="0" maxOccurs="unbounded"
      namespace="##any" processContents="lax"/>
  </xsd:sequence>
  <xsd:anyAttribute namespace="##any"/>
</xsd:complexType>
```

Example 10.1

A `complexType` construct containing fine and coarse-grained constraints.

As indicated by the bolded text, there are elements with specific names and data types that represent parts of the message definition with a *fine* level of constraint granularity. All the message instances (the actual XML documents that will be created based on this structure) must conform to these constraints to be considered valid (which is why these are considered the absolute "minimum" constraints).

The italicized text shows the element and attribute wildcards also contained by this complex type. These represent parts of the message definition with an extremely *coarse* level of constraint granularity in that messages do not need to comply to these parts of the message definition at all.

The use of the terms "fine-grained" and "coarse-grained" is highly subjective. What may be a fine-grained constraint in one contract may not be in another. The point is to understand how these terms can be applied when comparing parts of a message definition or when comparing different message definitions with each other.

10.2 Versioning and Compatibility

The number one concern when developing and deploying a new version of a service contract is the impact it will have on other parts of the enterprise that have formed or will form dependencies on it. This measure of impact is directly related to how compatible the new contract version is with the old version and its surroundings in general.

This section establishes the fundamental types of compatibility that relate to the content and design of new contract versions and also tie into the goals and limitations of different versioning strategies introduced at the end of this chapter.

Backwards Compatibility

A new version of a service contract that continues to support consumer programs designed to work with the old version is considered *backwards compatible*. From a design perspective, this means that the new contract has not changed in such a way that it can impact existing consumer programs that are already using the contract.

Backwards Compatibility in Web Services

Example 10.2 provides a simple instance of a backwards-compatible change based on the addition of a new operation to an existing WSDL definition:

```
<definitions name="Purchase Order" targetNamespace=
  "http://actioncon.com/contract/po"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:tns="http://actioncon.com/contract/po"
  xmlns:po="http://actioncon.com/schema/po">
  ...
  <portType name="ptPurchaseOrder">
    <operation name="opSubmitOrder">
      <input message="tns:msgSubmitOrderRequest"/>
      <output message="tns:msgSubmitOrderResponse"/>
    </operation>
    <operation name="opCheckOrderStatus">
      <input message="tns:msgCheckOrderRequest"/>
      <output message="tns:msgCheckOrderResponse"/>
    </operation>
    <operation name="opChangeOrder">
      <input message="tns:msgChangeOrderRequest"/>
      <output message="tns:msgChangeOrderResponse"/>
    </operation>
    <operation name="opCancelOrder">
      <input message="tns:msgCancelOrderRequest"/>
      <output message="tns:msgCancelOrderResponse"/>
    </operation>
    <operation name="opGetOrder">
      <input message="tns:msgGetOrderRequest"/>
      <output message="tns:msgGetOrderResponse"/>
    </operation>
  </portType>
</definitions>
```

Example 10.2
The addition of a new operation represents a common backwards-compatible change.

By adding a brand-new operation, we are creating a new version of the contract, but this change is backwards-compatible and will not impact any existing consumers. The new service implementation will continue to work with old service consumers because all the operations that an existing service consumer might invoke are still present and continue to meet the requirements of the previous service contract version.

Backwards Compatibility in REST Services

A backwards-compatible change to a REST-compliant service contract might involve adding some new resources or adding new capabilities to existing resources. In each of these cases the existing service consumers will only invoke the old methods on the old resources, which continue to work as they previously did.

As demonstrated in Example 10.3, supporting a new method that existing service consumers don't use results in a backwards-compatible change. However, in a service inventory with multiple REST services, we can take steps to ensure that new service consumers will continue to work with old versions of services.

```
Service: po.actioncon.com
Capabilities:
POST /orders
      In = application/vnd.com.actioncon.po+xml
GET /orders/{order-id}/status
      Out = text/plain
PUT /orders/{order-id}
      In = application/vnd.com/actioncon.po+xml
DELETE /orders/{order-id}
GET /orders/{order-id}
      Out = application/vnd.com.actioncon.po+xml
```

Example 10.3
The addition of a new resource or new supported method on a resource is a backwards-compatible change for a REST service.

As shown in Example 10.4, it may be important for service consumers to have a reasonable way of proceeding with their interaction if the service reports that the new method is not implemented.

```
Legal methods for actioncon.com service inventory:
* GET
* PUT
* DELETE
* POST
* SUBSCRIBE (consumers must fall back to periodic GET if service
reports "not implemented")
```

Example 10.4
New methods added to a service inventory's uniform contract need to provide a way for service consumers to "fall back" on a previously used method if they are to truly be backwards-compatible.

Changes to schemas and media types approach backwards compatibility in a different manner, in that they describe how information can be encoded for transport, and will often be used in both request and response messages. The focus for backwards compatibility is on whether a new message recipient can make sense of information sent by a legacy source. In other words, the new processor must continue to understand information produced by a legacy message *generator*.

An example of a change made to a schema for a message definition that is backwards-compatible is the addition of an optional element (as shown in bolded markup code in Example 10.5).

```
Media type = application/vnd.com.actioncon.po+xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://actioncon.com/schema/po"
  xmlns="http://actioncon.com/schema/po">
  <xsd:element name="LineItem" type="LineItemType"/>
  <xsd:complexType name="LineItemType">
    <xsd:sequence>
      <xsd:element name="productID" type="xsd:string"/>
      <xsd:element name="productName" type="xsd:string"/>
      <xsd:element name="available" type="xsd:boolean"
        minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Example 10.5

In an XML Schema definition, the addition of an optional element is also considered backwards-compatible.

Here we are using a simplified version of the XML Schema definition for the Purchase Order service. The optional `available` element is added to the `LineItemType` complex type. This has no impact on existing generators because they are not required to provide this element in their messages. New processors must be designed to cope without the new information if they are to remain backwards-compatible.

Changing any of the existing elements in the previous example from required to optional (by adding the `minOccurs="0"` setting) would also be considered a backwards-compatible change. When we have control over how we choose to design the next version of a Web service contract, backwards compatibility is generally attainable. However, mandatory changes (such as those imposed by laws or regulations) can often force us to break backwards compatibility.

NOTE

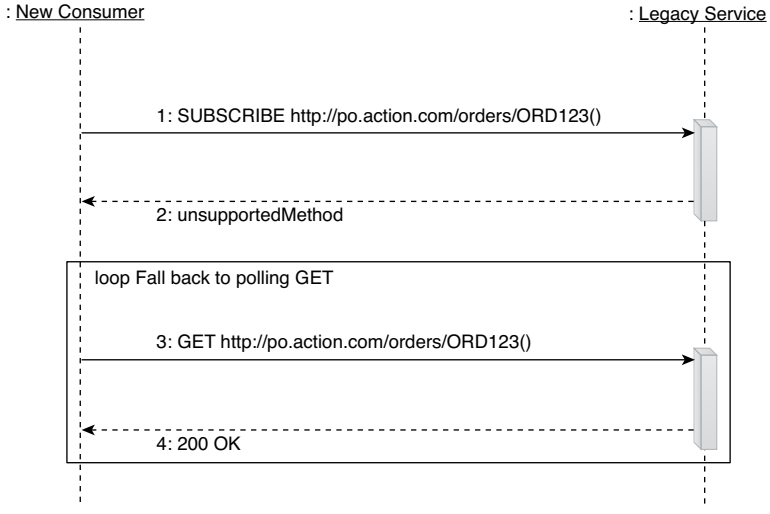
Both the Flexible and Loose versioning strategies explained at the end of this chapter support backwards compatibility.

Forwards Compatibility

When a service contract is designed in such a manner so that it can support a range of future consumer programs, it is considered to have an extent of *forwards compatibility*. This means that the contract can essentially accommodate how consumer programs will evolve over time.

Supporting forwards compatibility for Web service operations or uniform contract methods requires exception types to be present in the contract to allow service consumers to recover if they attempt to invoke a new and unsupported operation or method. For example, a "method not implemented" response enables the service consumer to detect that it is dealing with an incompatible service, thereby allowing it to handle this exception gracefully.

Redirection exception codes help REST services that implement a uniform contract change the resource identifiers in the contract when required. This is another way in which service contracts can allow legacy service consumers to continue using the service after contract changes have taken place (Example 10.6).



Example 10.6

A REST service ensures forwards compatibility by raising an exception whenever it does not understand a reusable contract or uniform contract method.

Forwards compatibility of schemas in REST services requires extension points to be present where new information can be added so that it will be safely ignored by legacy processors.

For example:

- Any validation that the processor does must not reject a document formatted according to the new schema.
- All existing information that the processor might need must remain present in future versions of the schema.
- Any new information added to the schema must be safe for legacy processors to ignore (if processors must understand the new information, then the change cannot be forwards compatible).
- The processor must ignore any information that it does not understand.

A common way to ensure validation does not reject future versions of the schema is to use wildcards in the earlier version. These provide extension points where new information can be added in future schema versions, as shown in Example 10.7.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://actioncon.com/schema/po"
  xmlns="http://actioncon.com/schema/po">
  <xsd:element name="LineItem" type="LineItemType"/>
  <xsd:complexType name="LineItemType">
    <xsd:sequence>
      <xsd:element name="productID" type="xsd:string"/>
      <xsd:element name="productName" type="xsd:string"/>
      <xsd:any namespace="##any" processContents="lax"
        minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:anyAttribute namespace="##any"/>
  </xsd:complexType>
</xsd:schema>
```

Example 10.7

To support forwards compatibility within a message definition generally requires the use of XML Schema wildcards.

In this example, the `xsd:any` and `xsd:anyAttribute` elements are added to allow for a range of unknown elements and data to be accepted by the service contract. In other words, the schema is being designed in advance to accommodate unforeseen changes in the future.

It is important to understand that building extension points into service contracts for forwards compatibility by no means eliminates the need to consider compatibility issues when making contract changes. New information can only be added to schemas

in a forwards-compatible manner if it is genuinely safe for processors to ignore. New operations are only able to be made forwards-compatible if a service consumer has an existing operation to fall back on when it finds the one it initially attempted to invoke is unsupported.

A service with a forwards-compatible contract will often not be able to process all message content. Its contract is simply designed to accept a broader range of data unknown at the time of its design.

NOTE

Forwards compatibility forms the basis of the Loose versioning strategy that is explained shortly.

Compatible Changes

When we make a change to a service contract that does not negatively affect existing consumers, then the change itself is considered a *compatible change*.

NOTE

In this book, the term "compatible change" refers to backwards compatibility by default. When used in reference to forwards compatibility, it is further qualified as a *forwards-compatible change*.

A simple example of a compatible change is when we set the `minOccurs` attribute of an element from "1" to "0," effectively turning a required element into an optional one, as shown in Example 10.8.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://actioncon.com/schema/po"
  xmlns="http://actioncon.com/schema/po">
  <xsd:element name="LineItem" type="LineItemType"/>
  <xsd:complexType name="LineItemType">
    <xsd:sequence>
      <xsd:element name="productID" type="xsd:string"/>
      <xsd:element name="productName" type="xsd:string"
        minOccurs="0"/>
      <xsd:element name="available" type="xsd:boolean"
        minOccurs="0"/>
```

```
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Example 10.8

The default value of the `minOccurs` attribute is "1". Therefore because this attribute was previously absent from the `productName` element declaration, it was considered a required element. Adding the `minOccurs="0"` setting turns it into an optional element, resulting in a compatible change. (Note that making this change to a message output from the service would be an incompatible change.)

This type of change will not impact existing consumer programs that are used to sending the element value to the Web service, nor will it affect future consumer programs that can be designed to optionally send that element.

Another example of a compatible change was provided earlier in Example 10.3, when we first added the optional `available` element declaration. Even though we extended the type with a whole new element, because it is optional it is considered a compatible change.

Here is a list of common compatible changes:

- Adding a new WSDL operation definition and associated message definitions
- Adding a new standard method to an existing REST resource
- Adding a set of new REST resources
- Changing the identifiers for a set of REST resources (including splitting and merging of services) using redirection response codes to facilitate migration of REST service consumers to the new identifiers
- Adding a new WSDL port type definition and associated operation definitions
- Adding new WSDL binding and service definitions
- Extending an existing uniform contract method in a way that can be safely ignored by REST services that can fall back on old service logic (for example, adding "If-None-Match" as a feature of the HTTP GET operation so that if the service ignores it, the consumer will still get the current and correct representation for the resource)
- Adding a new uniform contract method when an exception response exists for services that do not understand the method to use (and consumers can recover from this exception)

- Adding a new optional XML Schema element or attribute declaration to a message definition
- Reducing the constraint granularity of an XML Schema element or attribute of a message definition type used for input messages
- Adding a new XML Schema wildcard to a message definition type
- Adding a new optional WS-Policy assertion
- Adding a new WS-Policy alternative

Incompatible Changes

If after a change a contract is no longer compatible with consumers, then it is considered to have received an *incompatible change*. These are the types of changes that can break an existing contract and therefore impose the most challenges when it comes to versioning.

NOTE

The term "incompatible change" also indicates backwards compatibility by default. Incompatible changes that affect forwards compatibility will be qualified as "forwards-incompatible changes."

Going back to our example, if we set an element's `minOccurs` attribute from "0" to any number above zero, then we are introducing an incompatible change for input messages, as shown in Example 10.9:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://actioncon.com/schema/po"
  xmlns="http://actioncon.com/schema/po">
  <xsd:element name="LineItem" type="LineItemType"/>
  <xsd:complexType name="LineItemType">
    <xsd:sequence>
      <xsd:element name="productID" type="xsd:string"/>
      <xsd:element name="productName" type="xsd:string"
        minOccurs="3"/>
      <xsd:element name="available" type="xsd:boolean"
        minOccurs="3"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Example 10.9

Incrementing the `minOccurs` attribute value of any established element declaration is automatically an incompatible change.

What was formerly an optional element is now required. This will certainly affect existing consumers that are not designed to comply with this new constraint, because adding a new required element introduces a mandatory constraint upon the contract.

Common incompatible changes include:

- Renaming an existing WSDL operation definition
- Removing an existing WSDL operation definition
- Changing the MEP of an existing WSDL operation definition
- Adding a fault message to an existing WSDL operation definition
- Adding a new required XML Schema element or attribute declaration to a message definition
- Increasing the constraint granularity of an XML Schema element or attribute declaration of a message definition
- Renaming an optional or required XML Schema element or attribute in a message definition
- Removing an optional or required XML Schema element or attribute or wildcard from a message definition
- Adding a new required WS-Policy assertion or expression
- Adding a new ignorable WS-Policy expression (most of the time)

Incompatible changes tend to cause most of the challenges with service contract versioning.

10.3 REST Service Compatibility Considerations

REST services within a given service inventory typically share a uniform contract for every resource, including uniform methods and media types. The same media types are used in both requests and responses, and new uniform contract facets are reused much more often than they are added to. This emphasis on service contract reuse within REST-compliant service inventories results in the need to highlight some special considerations, because changes to the uniform contract will automatically impact a range of service consumers because:

- The uniform contract methods are shared by all services.
- The uniform contract media types are shared by both services and service consumers.

As a result, both backwards compatibility and forwards compatibility considerations are almost equally important.

SOA PATTERNS

Service contracts that make use of the Schema Centralization [356] pattern without necessarily being REST-compliant will often need to impose a similarly rigid view of forwards compatibility and backwards compatibility.

Uniform contract methods codify the kinds of interactions that can occur between services and their consumers. For example, GET codifies "fetch some data," while PUT codifies "store some data."

Because the kinds of interactions that occur between REST services within the same service inventory tend to be relatively limited and stable, methods will usually change at a low rate compared to media types or resources. Compatibility issues usually pertain to a set of allowable methods that are only changed after careful case-by-case consideration.

An example of a compatible change to HTTP is the addition of `If-None-Match` headers to GET requests. If a service consumer knows the last version (or `etag`) of the resource that it fetched, it can make its GET request conditional. The `If-None-Match` header allows the consumer to state that the GET request should not be executed if the version of the resource is still the same as it was for the consumer's last fetch. Instead, it will return the normal GET response, although it will do so in a non-optimal mode.

An example of an incompatible change to HTTP is the addition of a `Host` header used to support multihoming of Web servers. HTTP/1.0 did not require the name of the service to be included in request messages, but HTTP/1.1 does require this. If the special `Host` header is missing, HTTP/1.1 services must reject the request as being badly formed. However, HTTP/1.1 services are also required to be backwards-compatible, so if an HTTP/1.0 request comes into the REST service it will still be handled according to HTTP/1.0 rules.

Uniform contract media types further codify the kinds of information that can be exchanged between REST services and consumers. As previously stated, media types tend to change at a faster rate than HTTP methods in the uniform contract; however, media types still change more slowly than resources. Compatible change is more of a

live concern for the media types, and we can draw some more general rules about how to deal with them.

For example, if the generator of a message indicates to a processor of the message that it conforms to a particular media type, the processor generally does not need to know which version of the schema was used, nor does the processor need to have been built against the same version of the schema. The processor expects that all versions of the schema for a particular media type will be both forwards compatible and backwards-compatible with the type it was developed to support. Likewise, the generator expects that when it produces a message conformant with a particular schema version, that all processors of the message will understand it.

When incompatible changes are made to a schema, a new media type identifier is generally required to ensure that:

- The processor can decide how to parse a document based on the media type identifier
- Services and consumers are able to *negotiate* for a specific media type that will be understood by the processor when the message has been produced

Content negotiation is the ultimate fallback to ensure compatibility in REST-compliant service inventories. For a fetch interaction this often involves the consumer indicating to the service what media types it is able to support, and the service returning the most appropriate type that it supports. This mechanism allows for incompatible changes to be made to media types, as required.

NOTE

One way to better understand versioning issues that pertain to media types is to look at how they are used in HTML. An example of a compatible change to HTML that did not result in the need for a new media type was the addition of the `abbr` element to version 4.0 of the HTML language. This element allows new processors of HTML documents to support a mouse-over to expand abbreviations on a web page and to better support accessibility of the page. Legacy processors safely ignore the expansion, but will continue correctly showing the abbreviation itself.

An example of an incompatible change to HTML that did require a new media type was the conversion of HTML 4.0 to XML (resulting in version 1.0 of XHTML). The media type for the traditional SGML version remained `text/html`, while the XML version became `application/xhtml+xml`. This allowed content negotiation to occur between the two

types, and for processors to choose the correct parser and validation strategy based on which type was specified by the service.

Some incompatible changes have also been made to HTML without changing the media type. HTML 4.0 deprecated APPLET, BASEFONT, CENTER, DIR, FONT, ISINDEX, MENU, S, STRIKE, and U elements in favor of newer elements. These elements must continue to be understood but their use in HTML documents is being phased out. HTML 4.0 made LISTING, PLAINTEXT, and XMP obsolete. These elements should not be used in HTML 4.0 documents and no longer need to be understood.

Deprecating elements over a long period of time and eventually identifying them as obsolete once they are no longer used by existing services or consumers is a technique that can be used for REST media types to incrementally update a schema without having to change the media type.

10.4 Version Identifiers

One of the most fundamental design patterns related to Web service contract design is the Version Identification pattern. It essentially advocates that version numbers should be clearly expressed, not just at the contract level, but right down to the versions of the schemas that underlie the message definitions.

The first step to establishing an effective versioning strategy is to decide on a common means by which versions themselves are identified and represented within Web service contracts.

Versions are almost always communicated with version numbers. The most common format is a decimal, followed by a period and then another decimal, as shown here:

```
version="2.0"
```

Sometimes, you will see additional period plus decimal pairs that lead to more detailed version numbers like this:

```
version="2.0.1.1"
```

The typical meaning associated with these numbers is the measure or significance of the change. Incrementing the first decimal generally indicates a major version change (or upgrade) in the software, whereas decimals after the first period usually represent various levels of minor version changes.

From a compatibility perspective, we can associate additional meaning to these numbers. Specifically, the following convention has emerged in the industry:

- A minor version is expected to be backwards-compatible with other minor versions associated with a major version. For example, version 5.2 of a program should be fully backwards-compatible with versions 5.0 and 5.1.
- A major version is generally expected to break backwards compatibility with programs that belong to other major versions. This means that program version 5.0 is not expected to be backwards-compatible with version 4.0.

NOTE

A third "patch" version number is also sometimes used to express changes that are both forwards-compatible and backwards-compatible. Typically these versions are intended to clarify the schema only, or to fix problems with the schema that were discovered after it was deployed. For example, version 5.2.1 is expected to be fully compatible with version 5.2.0, but may be added for clarification purposes.

This convention of indicating compatibility through major and minor version numbers is referred to as the *compatibility guarantee*. Another approach, known as "amount of work," uses version numbers to communicate the effort that has gone into the change. A minor version increase indicates a modest effort, and a major version increase predictably represents a lot of work.

These two conventions can be combined and often are. The result is often that version numbers continue to communicate compatibility as explained earlier, but they sometimes increment by several digits, depending on the amount of effort that went into each version.

There are various syntax options available to express version numbers. For example, you may have noticed that the declaration statement that begins an XML document can contain a number that expresses the version of the XML specification being used:

```
<?xml version="1.0"?>
```

That same `version` attribute can be used with the root `xsd:schema` element, as follows:

```
<xsd:schema version="2.0" ...>
```

You can further create a custom variation of this attribute by assigning it to any element you define (in which case you are not required to name the attribute "version").

```
<LineItem version="2.0">
```

An alternative custom approach is to embed the major version number into a namespace or media type identifier, as shown here:

```
<LineItem xmlns="http://actioncon.com/schema/po/v2">
```

or

```
application/vnd.com.actioncon.po.v2+xml
```

Note that it has become a common convention to use date values in namespaces when versioning XML schemas, as follows:

```
<LineItem xmlns="http://actioncon.com/schema/po/2010/09">
```

In this case, it is the date of the change that acts as the major version identifier. To keep the expression of XML Schema definition versions in alignment with WSDL definition versions, we use version numbers instead of date values in upcoming examples. However, when working in an environment where XML Schema definitions are separately owned as part of an independent data architecture, it is not uncommon for schema versioning identifiers to be different from those used by WSDL definitions.

Regardless of which option you choose, it is important to consider the Canonical Versioning pattern that dictates that the expression of version information must be standardized across all service contracts within the boundary of a service inventory. In larger environments, this will often require a central authority that can guarantee the linearity, consistency, and description quality of version information. These types of conventions carry over into how service termination information is expressed, as further explored in Chapter 23 in *Web Service Contract Design and Versioning for SOA*.

SOA PATTERNS

Of course you may also be required to work with third-party schemas and WSDL definitions that may already have implemented their own versioning conventions. In this case, the extent to which the Canonical Versioning [327] pattern can be applied will be limited.

10.5 Versioning Strategies

There is no one versioning approach that is right for everyone. Because versioning represents a governance-related phase in the overall lifecycle of a service, it is a practice that is subject to the conventions, preferences, and requirements that are distinct to any enterprise.

Even though there is no de facto versioning technique for the WSDL, XML Schema, and WS-Policy content that comprises Web service contracts, a number of common and advocated versioning approaches have emerged, each with its own benefits and tradeoffs.

In this section, we single out the following three common strategies:

- *Strict* – Any compatible or incompatible changes result in a new version of the service contract. This approach does not support backwards or forwards compatibility.
- *Flexible* – Any incompatible change results in a new version of the service contract and the contract is designed to support backwards compatibility but not forwards compatibility.
- *Loose* – Any incompatible change results in a new version of the service contract and the contract is designed to support backwards compatibility and forwards compatibility.

These strategies are explained individually in the upcoming sections.

The Strict Strategy (New Change, New Contract)

The simplest approach to Web service contract versioning is to require that a new version of a contract be issued whenever any kind of change is made to any part of the contract.

This is commonly implemented by changing the target namespace value of a WSDL definition (and possibly the XML Schema definition) every time a compatible or incompatible change is made to the WSDL, XML Schema, or WS-Policy content related to the contract. Namespaces are used for version identification instead of a version attribute because changing the namespace value automatically forces a change in all consumer programs that need to access the new version of the schema that defines the message types.

This "super-strict" approach is not really that practical, but it is the safest and sometimes warranted when there are legal implications to Web service contract modifications, such as when contracts are published for certain interorganization data exchanges. Because both compatible and incompatible changes will result in a new contract version, this approach supports neither backwards nor forwards compatibility.

Pros and Cons

The benefit of this strategy is that you have full control over the evolution of the service contract, and because backwards and forwards compatibility are intentionally disregarded, you do not need to concern yourself with the impact of any change in particular (because all changes effectively break the contract).

On the downside, by forcing a new namespace upon the contract with each change, you are guaranteeing that all existing service consumers will no longer be compatible with any new version of the contract. Consumers will only be able to continue communicating with the Web service while the old contract remains available alongside the new version or until the consumers themselves are updated to conform to the new contract.

Therefore, this approach will increase the governance burden of individual services and will require careful transitioning strategies. Having two or more versions of the same service co-exist at the same time can become a common requirement for which the supporting service inventory infrastructure needs to be prepared.

The Flexible Strategy (Backwards Compatibility)

A common approach used to balance practical considerations with an attempt at minimizing the impact of changes to Web service contracts is to allow compatible changes to occur without forcing a new contract version, while not attempting to support forwards compatibility at all.

This means that any backwards-compatible change is considered safe in that it ends up extending or augmenting an established contract without affecting any of the service's existing consumers. A common example of this is adding a new operation to a WSDL definition or adding an optional element declaration to a message's schema definition.

As with the Strict strategy, any change that breaks the existing contract does result in a new contract version, usually implemented by changing the target namespace value of the WSDL definition and potentially also the XML Schema definition.

Pros and Cons

The primary advantage to this approach is that it can be used to accommodate a variety of changes while consistently retaining the contract's backwards compatibility. However, when compatible changes are made, these changes become permanent and cannot be reversed without introducing an incompatible change. Therefore, a governance process is required during which each proposed change is evaluated so that contracts do not become overly bloated or convoluted. This is an especially important consideration for agnostic services that are heavily reused.

The Loose Strategy (Backwards and Forwards Compatibility)

As with the previous two approaches, this strategy requires that incompatible changes result in a new service contract version. The difference here is in how service contracts are initially designed.

Instead of accommodating known data exchange requirements, special features from the WSDL, XML Schema, and WS-Policy languages are used to make parts of the contract intrinsically extensible so that they remain able to support a broad range of future, unknown data exchange requirements. For example:

- The `anyType` attribute value provided by the WSDL 2.0 language allows a message to consist of any valid XML document.
- XML Schema wildcards can be used to allow a range of unknown data to be passed in message definitions.
- Ignorable policy assertions can be defined to communicate service characteristics that can optionally be acknowledged by future consumers.

These and other features related to forwards compatibility are discussed in *Web Service Contract Design and Versioning for SOA*.

Pros and Cons

The fact that wildcards allow undefined content to be passed through Web service contracts provides a constant opportunity to further expand the range of acceptable message element and data content. On the other hand, the use of wildcards will naturally result in vague and overly coarse service contracts that place the burden of validation on the underlying service logic.

Strategy Summary

Provided in Table 10.1 is a broad summary of how the three strategies compare based on three fundamental characteristics.

	Strategy		
	Strict	*Flexible*	*Loose*
Strictness	High	Medium	Low
Governance Impact	High	Medium	High
Complexity	Low	Medium	High

Table 10.1
A general comparison of the three versioning strategies.

The three characteristics used in this table to form the basis of this comparison are as follows:

- *Strictness* – The rigidity of the contract versioning options. The Strict approach clearly is the most rigid in its versioning rules, while the Loose strategy provides the broadest range of versioning options due to its reliance on wildcards.
- *Governance Impact* – The amount of governance burden imposed by a strategy. Both Strict and Loose approaches increase governance impact but for different reasons. The Strict strategy requires the issuance of more new contract versions, which impacts surrounding consumers and infrastructure, while the Loose approach introduces the concept of unknown message sets that need to be separately accommodated through custom programming.
- *Complexity* – The overall complexity of the versioning process. Due to the use of wildcards and unknown message data, the Loose strategy has the highest complexity potential, while the straightforward rules that form the basis of the Strict approach make it the simplest option.

Throughout this comparison, the Flexible strategy provides an approach that represents a consistently average level of strictness, governance effort, and overall complexity.

10.6 REST Service Versioning Considerations

REST services that share the same uniform contract maintain separate versioned specifications for the following:

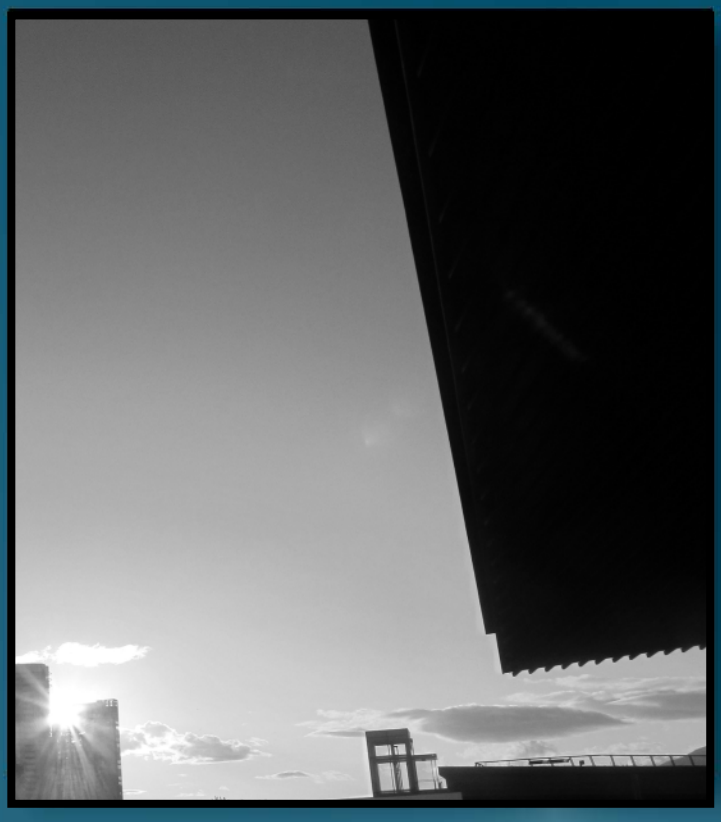
- The version number or specification of the resource identifier syntax (as per the "Request for Comments 6986 - Uniform Resource Identifier (URI): Generic Syntax" specification)
- The specification of the collection of legal methods, status codes, and other interaction protocol details (as per the "Request for Comments 2616 - Hypertext Transfer Protocol - HTTP/1.1" specification)
- Individual specifications for legal media types (for example. HTML 4.01 and the "Request for Comments 4287 - The Atom Syndication Format" specification)
- Individual specifications for service contracts that use the legal resource identifier syntax, methods, and media types

Each part of the uniform contract is specified and versioned independently of the others. Changing any one specification does not generally require another specification to be updated or versioned. Likewise, changing any of the uniform contract facet specifications does not require changes to individual service contracts, or changes to their version numbers.

This last point is in contradiction to some conventional versioning strategies. One might expect that if a schema used in a service contract changed, then the service contract would need to be modified. However, with REST services there is a tendency to maintain both forwards compatibility and backwards compatibility. If a REST service consumer sends a message that conforms to a newer schema, the service can process it as if it conformed to the older schema. If compatibility between these schemas has been maintained, then the service will function correctly. Likewise, if the service returns a message to the consumer that conforms to an old schema, the newer service consumer can still process the message correctly.

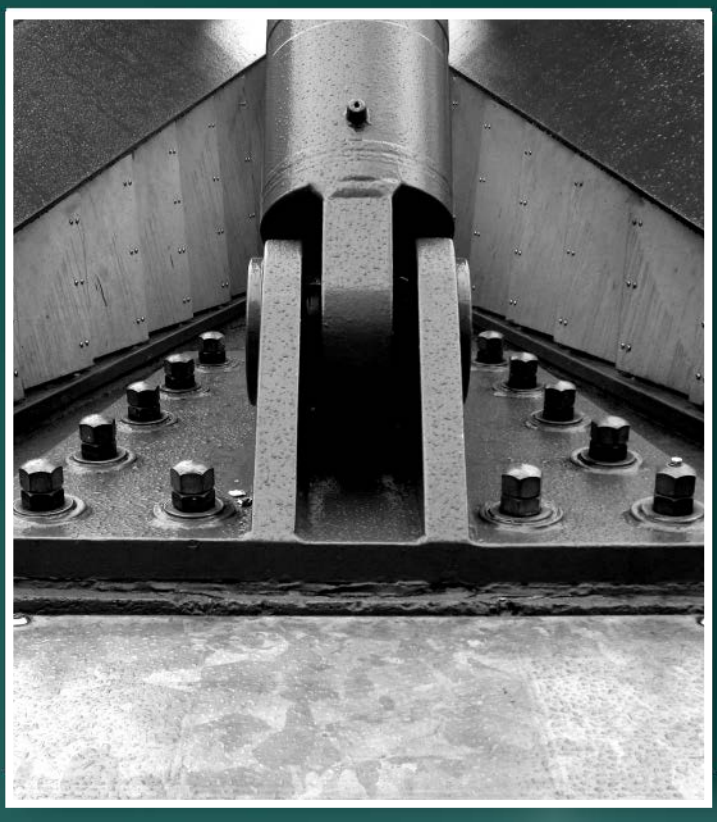
REST service contracts only need to directly consider the versioning of the uniform contract when media types used become deprecated, or when the schema advances so far that elements and attributes the service depends on are on their way to becoming obsolete. When this occurs, the service contract needs to be updated, and with it, the underlying service logic that processes the media types.

Part III



Appendices

This page intentionally left blank



Appendix A

Service-Orientation Principles Reference

This appendix provides profile tables for the service-orientation principles referenced throughout this book. As explained in Chapter 1, each principle reference is suffixed with the page number of its corresponding profile table in this appendix.

Every profile table contains the following sections:

- *Short Definition* – A concise, single-statement definition that establishes the fundamental purpose of the principle.
- *Long Definition* – A longer description of the principle that provides more detail as to what it is intended to accomplish.
- *Goals* – A list of specific design goals that are expected from the application of the principle. Essentially, this list provides the ultimate results of the principle's realization.
- *Design Characteristics* – A list of specific design characteristics that can be realized via the application of the principle. This provides some insight as to how the principle ends up shaping the service.
- *Implementation Requirements* – A list of common prerequisites for effectively applying the design principle. These can range from technology to organizational requirements.

Note that these profile tables provide only summarized versions of the principles. Complete coverage of the eight service-orientation design principles, including case studies, is provided in the *SOA Principles of Service Design* book.

For more information about this and other titles in the *Prentice Hall Service Technology Series from Thomas Erl*, visit www.servicetechbooks.com. Summarized content of topics related to service-orientation can also be found online at www.serviceorientation.com.

Standardized Service Contract	
Short Definition	*"Services share standardized contracts."*
Long Definition	*"Services within the same service inventory are in compliance with the same contract design standards."*
Goals	• To enable services with a meaningful level of natural interoperability within the boundary of a service inventory. This reduces the need for data transformation because consistent data models are used for information exchange. • To allow the purpose and capabilities of services to be more easily and intuitively understood. The consistency with which service functionality is expressed through service contracts increases interpretability and the overall predictability of service endpoints throughout a service inventory. Note that these goals are further supported by other service-orientation principles as well.
Design Characteristics	• A service contract (comprised of a technical interface or one or more service description documents) is provided with the service. • The service contract is standardized through the application of design standards.
Implementation Requirements	The fact that contracts need to be standardized can introduce significant implementation requirements to organizations that do not have a history of using standards. For example: • Design standards and conventions need to ideally be in place prior to the delivery of any service in order to ensure adequately scoped standardization. (For those organizations that have already produced ad-hoc Web services, retro-fitting strategies may need to be employed.) • Formal processes need to be introduced to ensure that services are modeled and designed consistently, incorporating accepted design principles, conventions, and standards.

	• Because achieving standardized service contracts generally requires a "contract-first" approach to service-oriented design, the full application of this principle will often demand the use of development tools capable of importing a customized service contract without imposing changes. • Appropriate skill sets are required to carry out the modeling and design processes with the chosen tools. When working with Web services, the need for a high level of proficiency with XML schema and WSDL languages is practically unavoidable. WS-Policy expertise may also be required. These and other requirements can add up to a noticeable transition effort that goes well beyond technology adoption.

Table A.1
A profile for the Standardized Service Contract principle

Service Loose Coupling	
Short Definition	*"Services are loosely coupled."*
Long Definition	*"Service contracts impose low consumer coupling requirements and are themselves decoupled from their surrounding environment."*
Goals	By consistently fostering reduced coupling within and between services, we are working toward a state where service contracts increase independence from their implementations and services are increasingly independent from each other. This promotes an environment in which services and their consumers can be adaptively evolved over time with minimal impact on each other.
Design Characteristics	• The existence of a service contract that is ideally decoupled from technology and implementation details. • A functional service context that is not dependent on outside logic. • Minimal consumer coupling requirements.
Implementation Requirements	• Loosely coupled services are typically required to perform more runtime processing than if they were more tightly coupled. As a result, data exchange in general can consume more runtime resources, especially during concurrent access and high usage scenarios. • Achieving the right balance of coupling, while also supporting the other service-orientation principles that affect contract design, requires increased service contract design proficiency.

Table A.2
A profile for the Service Loose Coupling principle

Service Abstraction	
Short Definition	*"Non-essential service information is abstracted."*
Long Definition	*"Service contracts only contain essential information and information about services is limited to what is published in service contracts."*
Goals	Many of the other principles emphasize the need to publish *more* information in the service contract. The primary role of this principle is to keep the quantity and detail of contract content concise and balanced and prevent unnecessary access to additional service details.
Design Characteristics	• Services consistently abstract specific information about technology, logic, and function away from the outside world (the world outside of the service boundary). • Services have contracts that concisely define interaction requirements and constraints and other required service meta details. • Outside of what is documented in the service contract, information about a service is controlled or altogether hidden within a particular environment.
Implementation Requirements	The primary prerequisite to achieving the appropriate level of abstraction for each service is the level of service contract design skill applied.

Table A.3
A profile for the Service Abstraction principle

Service Reusability	
Short Definition	*"Services are reusable."*
Long Definition	*"Services contain and express agnostic logic and can be positioned as reusable enterprise resources."*
Goals	The goals behind Service Reusability are tied directly to some of the most strategic objectives of service-oriented computing: • To allow for service logic to be repeatedly leveraged over time so as to achieve an increasingly high return on the initial investment of delivering the service. • To increase business agility on an organizational level by enabling the rapid fulfillment of future business automation requirements through wide-scale service composition. • To enable the realization of agnostic service models. • To enable the creation of service inventories with a high percentage of agnostic services.
Design Characteristics	• The logic encapsulated by the service is associated with a context that is sufficiently agnostic to any one usage scenario so as to be considered reusable. • The logic encapsulated by the service is sufficiently generic, allowing it to facilitate numerous usage scenarios by different types of service consumers. • The service contract is flexible enough to process a range of input and output messages. • Services are designed to facilitate simultaneous access by multiple consumer programs.

Implementation Requirements	From an implementation perspective, Service Reusability can be the most demanding of the principles we've covered so far. Below are common requirements for creating reusable services and supporting their long-term existence: • A scalable runtime hosting environment capable of high-to-extreme concurrent service usage. Once a service inventory is relatively mature, reusable services will find themselves in an increasingly large number of compositions. • A solid version control system to properly evolve contracts representing reusable services. • Service analysts and designers with a high degree of subject matter expertise who can ensure that the service boundary and contract accurately represent the service's reusable functional context. • A high level of service development and commercial software development expertise so as to structure the underlying logic into generic and potentially decomposable components and routines. These and other requirements place an emphasis on the appropriate staffing of the service delivery team, as well as the importance of a powerful and scalable hosting environment and supporting infrastructure.

Table A.4
A profile for the Service Reusability principle

Service Autonomy	
Short Definition	*"Services are autonomous."*
Long Definition	*"Services exercise a high level of control over their underlying runtime execution environment."*
Goals	• To increase a service's runtime reliability, performance, and predictability, especially when being reused and composed. • To increase the amount of control a service has over its runtime environment. By pursuing autonomous design and runtime environments, we are essentially aiming to increase post-implementation control over the service and the service's control over its own execution environment.
Design Characteristics	• Services have a contract that expresses a well-defined functional boundary that should not overlap with other services. • Services are deployed in an environment over which they exercise a great deal (and preferably an exclusive level) of control. • Service instances are hosted by an environment that accommodates high concurrency for scalability purposes.
Implementation Requirements	• A high level of control over how service logic is designed and developed. Depending on the level of autonomy being sought, this may also involve control over the supporting data models. • A distributable deployment environment, so as to allow the service to be moved, isolated, or composed as required. • An infrastructure capable of supporting desired autonomy levels.

Table A.5
A profile for the Service Autonomy principle

Service Statelessness	
Short Definition	*"Services minimize statefulness."*
Long Definition	*"Services minimize resource consumption by deferring the management of state information when necessary."*
Goals	• To increase service scalability. • To support the design of agnostic service logic and improve the potential for service reuse.
Design Characteristics	What makes this somewhat of a unique principle is the fact that it is promoting a condition of the service that is temporary in nature. Depending on the service model and state deferral approach used, different types of design characteristics can be implemented. Some examples include: • Highly business process-agnostic logic so that the service is not designed to retain state information for any specific parent business process. • Less constrained service contracts so as to allow for the receipt and transmission of a wider range of state data at runtime. • Increased amounts of interpretive programming routines capable of parsing a range of state information delivered by messages and responding to a range of corresponding action requests.
Implementation Requirements	Although state deferral can reduce the overall consumption of memory and system resources, services designed with statelessness considerations can also introduce some performance demands associated with the runtime retrieval and interpretation of deferred state data. Here is a short checklist of common requirements that can be used to assess the support of stateless service designs by vendor technologies and target deployment locations: • The runtime environment should allow for a service to transition from an idle state to an active processing state in a highly efficient manner.

	• Enterprise-level or high-performance XML parsers and hardware accelerators (and SOAP processors) should be provided to allow services implemented as Web services to more efficiently parse larger message payloads with less performance constraints. • The use of attachments may need to be supported by Web services to allow messages to include bodies of payload data that do not undergo interface-level validation or translation to local formats. The nature of the implementation support required by the average stateless service in an environment will depend on the state deferral approach used within the service-oriented architecture.

Table A.6
A profile for the Service Statelessness principle

Service Discoverability	
Short Definition	*"Services are discoverable."*
Long Definition	*"Services are supplemented with communicative metadata by which they can be effectively discovered and interpreted."*
Goals	• Services are positioned as highly discoverable resources within the enterprise. • The purpose and capabilities of each service are clearly expressed so that they can be interpreted by humans and software programs. Achieving these goals requires foresight and a solid understanding of the nature of the service itself. Depending on the type of service model being designed, realizing this principle may require both business and technical expertise.
Design Characteristics	• Service contracts are equipped with appropriate metadata that will be correctly referenced when discovery queries are issued. • Service contracts are further outfitted with additional meta information that clearly communicates their purpose and capabilities to humans. • If a service registry exists, registry records are populated with the same attention to meta information as just described. • If a service registry does not exist, service profile documents are authored to supplement the service contract and to form the basis for future registry records.

Implementation Requirements	• The existence of design standards that govern the meta information used to make service contracts discoverable and interpretable, as well as guidelines for how and when service contracts should be further supplemented with annotations. • The existence of design standards that establish a consistent means of recording service meta information outside of the contract. This information is either collected in a supplemental document in preparation for a service registry, or is placed in the registry itself. You may have noticed the absence of a service registry on the list of implementation requirements. As previously established, the goal of this principle is to implement design characteristics within the service, not within the architecture.

Table A.7
A profile for the Service Discoverability principle

Service Composability	
Short Definition	*"Services are composable."*
Long Definition	*"Services are effective composition participants, regardless of the size and complexity of the composition."*
Goals	When discussing the goals of Service Composability, most of the goals of Service Reusability apply. This is because service composition often turns out to be a form of service reuse. In fact, you may recall that one of the objectives we listed for the Service Reusability principle was to enable wide-scale service composition. However, above and beyond simply attaining reuse, service composition provides the medium through which we can achieve what is often classified as the ultimate goal of service-oriented computing. By establishing an enterprise comprised of solution logic represented by an inventory of highly reusable services, we provide the means for a large extent of future business automation requirements to be fulfilled through service composition.
Design Characteristics For Composition Member Capabilities	Ideally, every service capability (especially those providing reusable logic) is considered a potential composition member. This essentially means that the design characteristics already established by the Service Reusability principle are equally relevant to building effective composition members. Additionally, there are two further characteristics emphasized by this principle: • The service needs to possess a highly efficient execution environment. More so than being able to manage concurrency, the efficiency with which composition members perform their individual processing should be highly tuned. • The service contract needs to be flexible so that it can facilitate different types of data exchange requirements for similar functions. This typically relates to the ability of the contract to exchange the same type of data at different levels of granularity. The manner in which these qualities go beyond mere reuse has to do primarily with the service being capable of optimizing its runtime processing responsibilities in support of multiple, simultaneous compositions.

Design Characteristics for Composition Controller Capabilities	Composition members will often also need to act as controllers or sub-controllers within different composition configurations. However, services designed as designated controllers are generally alleviated from many of the high-performance demands placed on composition members. These types of services therefore have their own set of design characteristics: • The logic encapsulated by a designated controller will almost always be limited to a single business task. Typically, the task service model is used, resulting in the common characteristics of that model being applied to this type of service. • While designated controllers may be reusable, service reuse is not usually a primary design consideration. Therefore, the design characteristics fostered by Service Reusability are considered and applied where appropriate, but with less of the usual rigor applied to agnostic services. • Statelessness is not always as strictly emphasized on designated controllers as with composition members. Depending on the state deferral options available by the surrounding architecture, designated controllers may sometimes need to be designed to remain fully stateful while the underlying composition members carry out their respective parts of the overall task. Of course, any capability acting as a controller can become a member of a larger composition, which brings the previously listed composition member design characteristics into account as well.

Table A.8
A profile for the Service Composability principle

This page intentionally left blank

Appendix B

REST Constraints Reference

This appendix provides profile tables for the REST constraints referenced throughout this book. As explained in Chapter 1, each constraint reference is suffixed with the page number of its corresponding profile table in this appendix.

Every profile table contains the following sections:

- *Short Definition* – A concise, single-statement definition that establishes the fundamental purpose of the constraint.
- *Long Definition* – A longer description of the constraint that provides more detail as to what it is intended to accomplish.
- *Application* – A list of common steps and requirements for applying the constraint.
- *Impacts* – A list of positive and negative impacts that can result from the application of the constraint.
- *Relationship to REST* – A brief explanation of how the constraint can relate to other constraints and overall REST architecture.
- *Related REST Goals* – A list of REST design goals that are related to and relevant to the application of this constraint.
- *Related Service-Orientation Principles* – A list of service-orientation principles related to the constraint.
- *Related SOA Patterns* – A list of SOA design patterns related to the constraint.

Note that these profile tables provide only summarized versions of the constraints. Complete coverage of the REST constraints, including case studies, is provided in the *SOA with REST: Principles, Patterns & Constraints for Building Enterprise Solutions with REST* book.

For more information about this and other titles in the *Prentice Hall Service Technology Series from Thomas Erl*, visit www.servicetechbooks.com. Summarized content of REST-related topics can also be found online at www.whatisrest.com.

Client-Server	
Short Definition	*"Solution logic is separated into consumer and service logic that share a technical contract."*
Long Definition	*"Business automation logic is organized into a solution comprised of units of consumer and service logic. Service consumers actively invoke service capabilities by sending messages that comply with a published technical service contract. Services passively wait to process request messages and respond to their receipt in compliance with the technical contract."*
Application	• Solution logic must undergo a process whereby it is subjected to the separation of concerns. This partitions the logic into units that address defined concerns. These units of logic are composed to form the solution at runtime. • Thec onsumer's required knowledge about a service and the service's required knowledge of its consumers are limited to the contents of the shared technical contract.
Impacts	• Service logic can become more scalable and reusable because it is freed from having to implement consumer-specific logic. • Service and consumer logic are simplified due to respective information hiding. • Service and consumer implementations can be evolved independently in ways that do not require alterations to the shared contract. • Interactions between services and consumers that circumvent the shared technical contract are prohibited, potentially resulting in lost opportunities to optimize the solution architecture.
Relationship to REST	This is a foundational constraint that defines the separation between service, consumer, and the technical contract they share. All of the other constraints reference these artifacts and so build upon this constraint.
Related REST Goals	Modifiability, Scalability
Related Service-Orientation Principles	Service Loose Coupling (293), Service Abstraction (294)
Related SOA Patterns	Capability Composition [328], Contract Denormalization [335], Decoupled Contract [337], Functional Decomposition [344]

Stateless	
Short Definition	*"Services remain stateless between request/response message exchanges with service consumers."*
Long Definition	*"The communication between a service and a consumer is regulated so that the consumer provides all data necessary for the service to understand each consumer request. Between requests, the service is not permitted to retain any state data specific to its interaction with the consumer instance, allowing it to exist in a stateless condition. Instead, session state is deferred to the consumer at the end of each request."*
Application	• Consumer logic must be designed to preserve state data between requests and to issue request messages containing state data. • The request message must contain all of the state data necessary for the service to process the request, and the service must be able to "forget" the state data upon issuing the response without compromising the overall interaction. • Because the service is only involved in the automation of a solution when a consumer is actively making a request to it, in-between requests the service is "at rest," and therefore using no CPU, memory, or network resources on behalf of the consumer. • The service cannot be required to store data specific to a runtime instance of a service consumer. However, the service is still allowed to store data that is related to its own functional context.

Impacts	• Making consumers responsible for preserving state data alleviates the service from having to store and replicate potentially volatile data that is only relevant to the individual consumer program. • Deferring session state to consumers between requests frees up service memory resources, allowing the service to scale with the number of concurrent requests, rather than with the total number of concurrent consumers. • Messages can be understood by the service without the need to have inspected earlier messages. This can simplify service logic design and further reduce the complexity of debugging. • The requirement to repeatedly transmit potentially redundant state data can increase network traffic and processing overhead. • Reliability of state data can be both positively and negatively impacted: Service instance failures can be dealt with gracefully because the service does not retain state, but failure of the service consumer can result in a loss of state data.
Relationship to REST	While this constraint builds upon Client-Server {307}, it helps enable Cache {310} and Layered System {313}.
Related REST Goals	Modifiability, Scalability, Performance (negative), Visibility, Reliability
Related Service-Orientation Principles	Service Statelessness (298)
Related SOA Patterns	State Messaging [362]

Cache	
Short Definition	*"Service consumers can cache and reuse response message data."*
Long Definition	*"The data provided by a prior response message can be temporarily stored and reused by the service consumer for later request messages."*
Application	• Services must be designed to produce accurate cache control metadata and return it in response messages. Response messages are marked as cacheable or non-cacheable, either with explicit message metadata or as part of the contract definition. • An optional consumer-side or intermediary cache repository enables the consumer to reuse cacheable response data for later request messages. • Request messages must be comparable to determine whether or not they are equivalent. • Contracts must either include explicit statements about the cacheability of responses, or must allow for cache control metadata to be included in responses.
Impacts	• Runtimee fficiency is improved by eliminating the need for duplicate response messages to be transmitted and processed. • The cache provides a robust and simple mechanism to perform "lazy replication" of service state data to its consumers. • Some forms of cached data can become stale and outdated if not regularly checked and updated.
Relationship to REST	A number of established techniques for pushing data out to consumers are disallowed by the application of Client-Server {307} and Stateless {308}. The Cache constraint provides a mechanism that is permitted by other constraints and one that results in a simple and robust architecture for reusing and optimizing the distribution of data.
Related REST Goals	Performance, Scalability, Reliability (negative)
Related Service-Orientation Principles	n/a
Related SOA Patterns	State Messaging [362]

Uniform Contract	
Short Definition	*"Service consumers and services share a common, overarching, generic technical contract."*
Long Definition	*"Consumers access service capabilities via methods, media types, and a common resource identifier syntax that are standardized across many consumers and services. Service capabilities provide access to resources that can further provide links to other resources."*
Application	• A uniform contract with generic and reusable methods, media types, and resource identifier syntax is established for a collection of consumers and services. • Consumer message processing logic is designed to be tightly coupled to the uniform contract. • Consumer message processing logic is designed to be decoupled or loosely coupled to service-specific capabilities and resources. • Resources can further provide links to other resources that the service consumer can "discover" and optionally access, dynamically at runtime.
Impacts	• The application of this constraint results in baseline standardization of technical interface characteristics across all services within the scope of application. This level of standardization can foster interoperability across all affected services. • Standardization resulting from Uniform Contract can include canonical schemas associated with media types. The common use of such schemas can further improve the extent of intrinsic interoperability. • By limiting coupling to the uniform contract and leveraging dynamic binding, consumers and services can achieve reduced levels of overall coupling requirements. • It can be difficult to identify and entirely rely on built-in uniform contract semantics for machine-to-machine interactions that need to be reusable by multiple services and their consumers. • Request and response messages based on uniform methods and media types may contain more information than is strictly required for a particular interaction. The transfer of redundant data can increase performance overhead.

Relationship to REST	The Uniform Contract constraint builds upon Client-Server {307} to support reuse and composition of consumers and services.
Related REST Goals	Simplicity, Modifiability, Performance (negative), Visibility
Related Service-Orientation Principles	Standardized Service Contract (291), Service Loose Coupling (293), Service Abstraction (294)
Related SOA Patterns	Decoupled Contract [337]

Layered System	
Short Definition	*"A solution can be comprised of multiple architectural layers."*
Long Definition	*"A solution is defined in terms of architectural layers, where no one layer can see past the next. Layers can be comprised of consumers and services with published contracts or event-driven middleware components (intermediaries) that establish processing layers between consumers and services. In either case, logic within a given solution layer cannot have knowledge beyond the immediate layers above or below it (within the solution hierarchy)."*
Application	• Consumers are designed to invoke services without knowledge of what other services those services may also invoke. • Intermediaries are added to perform runtime message processing without knowledge of how those messages may be further processed beyond the next layer of processing. • The solution architecture is designed to allow new middleware layers to be added or old middleware layers to be removed without changing the technical contract between services and consumers. • Request and response messages must not reveal which layer the message comes from to their recipients.
Impacts	• At the consumer/service level, this constraint ensures an extent of information hiding, which naturally reduces consumer-to-service coupling. • At the middleware component level, this constraint advocates the use of cross-cutting agents capable of performing generic, utility-centric functions on messages exchanged by consumers and services. • These types of architectural layers can provide a flexible means of evolving a solution architecture and/or its underlying infrastructure while minimizing the impact on the solution logic itself. • The increased separation and distribution of moving parts performing solution logic processing can negatively impact the overall performance overhead (especially when middleware components are being reused by multiple solutions). • By limiting knowledge of the entire solution architecture to consumer designers, opportunities for optimizing the runtime performance of a solution can be lost.

Relationship to REST	The middleware components commonly introduced by the application of this constraint can directly support or enable Uniform Contract {311}, Cache {310}, and Stateless {308}.
Related REST Goals	Modifiability, Scalability, Performance (negative), Simplicity, Visibility
Related Service-Orientation Principles	Service Loose Coupling (293), Service Abstraction (294)
Related SOA Patterns	Capability Composition [328], Service Agent [357]

Code-on-Demand	
Short Definition	*"Service consumers support the execution of deferred service logic."*
Long Definition	*"Service consumer architectures include an execution environment for logic provided by a service. This deferred logic can be used to extend the functionality of the consumer, or to temporarily specialize it for a particular purpose."*
Application	• Service consumers are designed to process logic offloaded to them by services at runtime. • Services make explicit decisions as to whether they will execute logic themselves or defer the execution of that logic to their consumers.
Impacts	• Features can be dynamically added to consumers without the need for them to be formally upgraded. • Services are able to avoid becoming execution bottlenecks by deferring logic to consumers rather than executing the logic themselves. • The required execution environments for consumers to process service logic can introduce security vulnerabilities.
Relationship to REST	n/a
Related REST Goals	Modifiability, Scalability, Performance, Visibility (negative), Simplicity (negative)
Related Service-Orientation Principles	n/a
Related SOA Patterns	n/a